Sincronização e Comunicação entre Processos

Sincronização e Comunicação entre Processos

- Introdução
  - Década de 1960
    - Surgimento sistemas multiprogramáveis
    - Aplicações concorrentes
      - Compartilhamento de recursos
      - Possíveis situações indesejáveis
      - Necessidade de mecanismos de sincronização
  - Nesta unidade
    - Problemas de compartilhamento,
    - Sincronização, semáforos, monitores, *deadlock*

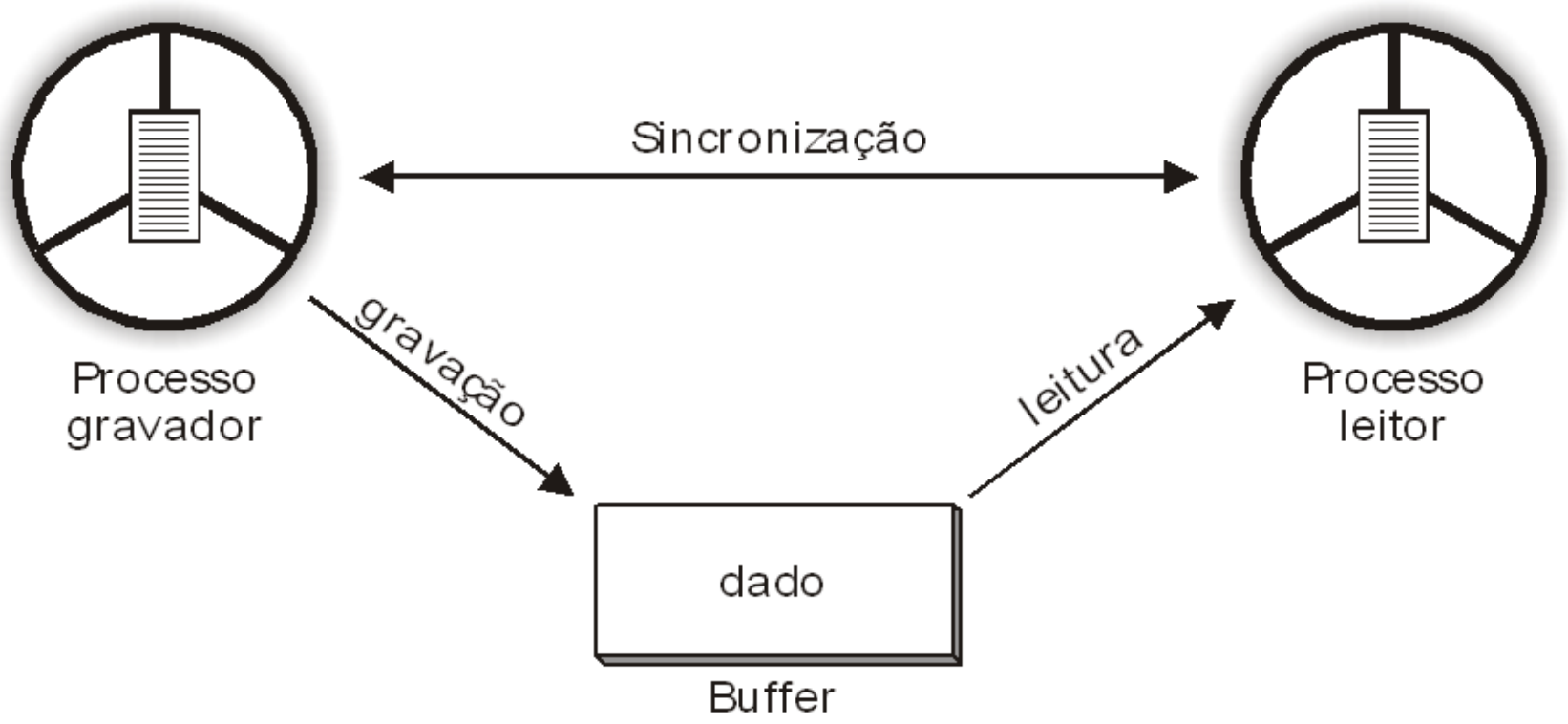
Aplicações concorrentes

- Processos se comunicam
  - Por meio de compartilhamento de memória
  - Por troca de mensagens

- Mecanismos de sincronização
  - Controlam a comunicação entre processos
  - Garantem a confiabilidade das aplicações
  - Processos, subprocessos ou threads usam os mesmos conceitos para sincronização

Aplicações concorrentes

- Sincronização e Comunicação entre Processos



Especificação de concorrência

- Existem várias notações
  - FORK e JOIN (Conway, 1963)

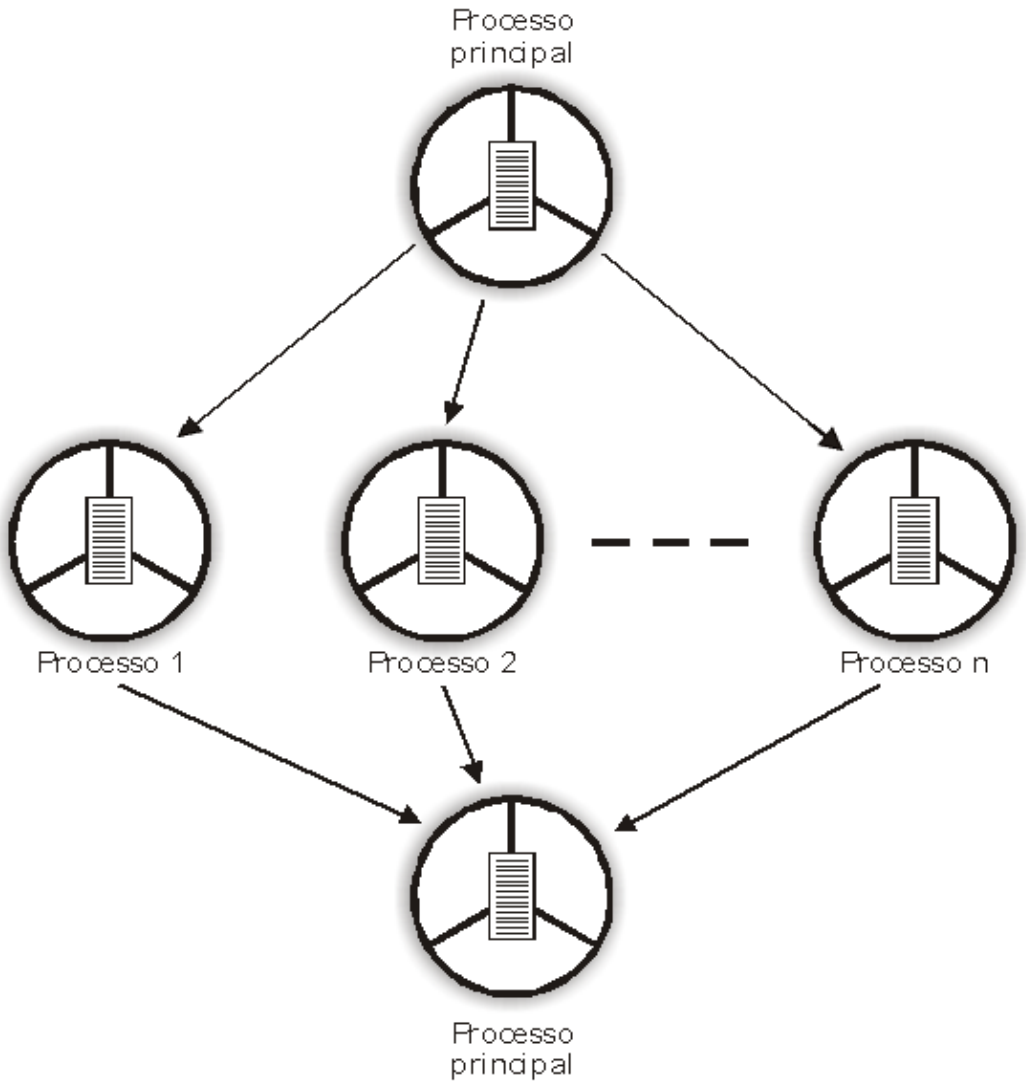
```
PROGRAM A;      PROGRAM B;
.               .
.               .
FORK B;         .
.               .
.               .
JOIN B;                 END.
.
END.
```

Especificação de concorrência

- PARBEGIN e PAREND (Dijkstra, 1965)
  - Comandos usados para sincronização
  - PARBEGIN
    - Especifica seqüência de comandos que serão executados em ordem imprevisível.
  - PAREND
    - Define um ponto de sincronização

Especificação de concorrência

```
PARBEGIN
    Comando_1;
    Comando_2;
       .
       .
    Comando_n;
PAREND
```



Especificação de concorrência

- X := SQRT(1024) + (35.4 * 0.23) – (302 / 7)

```
PROGRAM Expressao;
      VAR X, Temp1, Temp2, Temp3: REAL;
BEGIN
      PARBEGIN
            Temp1 := SQRT(1024);
            Temp2 := 35.4 * 0.23;
            Temp3 := 302 / 7;
      PAREND;
      X := Temp1 + Temp2 - Temp3;
      WRITELN('X = ', X);
END
```

Problemas de Compartilhamento

- Problema 1:
  - Compartilhamento de arquivos
  - Dois caixas de um banco tentam atualizar o saldo de um cliente ao mesmo tempo

```
PROGRAM Conta_Corrente;
  ...
  READ (Arq_Contas, Reg_Cliente);
  READLN (Valor_Dep_Ret);
  Reg_Cliente.Saldo := Reg_Cliente.Saldo+Valor_Dep_Ret;
  WRITE (Arq_Contas, Reg_Cliente);
  ...
END
```

Problemas de Compartilhamento

- Cenário 1:
  - Conta do cliente é movimentada por duas pessoas.
    - Saldo inicial é de R$1000,00
  - Processo do Caixa 1:
    - Debita R$200,00
  - Processo do Caixa 2:
    - Credita R$300,00
  - Qual o saldo esperado para a conta após as alterações?

Problemas de Compartilhamento

- Problema 1:
  - Processo do Caixa 1:
    - Lê registro
    - Debita valor na memória
  - Processo do Caixa 2
    - Lê registro antes de Caixa 1 gravar no arquivo
    - Credita valor na memória

Problemas de Compartilhamento

Caixa	*Comando*	*Saldo Arq*	*Valor C/D*	*Saldo Men*
Cx 1	READ	1.000	*	1.000

Problemas de Compartilhamento

Caixa	*Comando*	*Saldo Arq*	*Valor C/D*	*Saldo Men*
Cx 1	READ	1.000	*	1.000
Cx 1	READLN	1.000	-200	1.000

Problemas de Compartilhamento

Caixa	*Comando*	*Saldo Arq*	*Valor C/D*	*Saldo Men*
Cx 1	READ	1.000	*	1.000
Cx 1	READLN	1.000	-200	1.000
Cx 1	:=	1.000	-200	800

Problemas de Compartilhamento

Caixa	*Comando*	*Saldo Arq*	*Valor C/D*	*Saldo Men*
Cx 1	READ	1.000	*	1.000
Cx 1	READLN	1.000	-200	1.000
Cx 1	:=	1.000	-200	800
Cx 2	READ	1.000	*	1.000

Problemas de Compartilhamento

Caixa	*Comando*	*Saldo Arq*	*Valor C/D*	*Saldo Men*
Cx 1	READ	1.000	*	1.000
Cx 1	READLN	1.000	-200	1.000
Cx 1	:=	1.000	-200	800
Cx 2	READ	1.000	*	1.000
Cx 2	READLN	1.000	+300	1.000

Problemas de Compartilhamento

Caixa	*Comando*	*Saldo Arq*	*Valor C/D*	*Saldo Men*
Cx 1	READ	1.000	*	1.000
Cx 1	READLN	1.000	-200	1.000
Cx 1	:=	1.000	-200	800
Cx 2	READ	1.000	*	1.000
Cx 2	READLN	1.000	+300	1.000
Cx 2	:=	1.000	+300	1.300

Problemas de Compartilhamento

Caixa	*Comando*	*Saldo Arq*	*Valor C/D*	*Saldo Men*
Cx 1	READ	1.000	*	1.000
Cx 1	READLN	1.000	-200	1.000
Cx 1	:=	1.000	-200	800
Cx 2	READ	1.000	*	1.000
Cx 2	READLN	1.000	+300	1.000
Cx 2	:=	1.000	+300	1.300
Cx 1	WRITE	800	-200	800

Problemas de Compartilhamento

Caixa	*Comando*	*Saldo Arq*	*Valor C/D*	*Saldo Men*
Cx 1	READ	1.000	*	1.000
Cx 1	READLN	1.000	-200	1.000
Cx 1	:=	1.000	-200	800
Cx 2	READ	1.000	*	1.000
Cx 2	READLN	1.000	+300	1.000
Cx 2	:=	1.000	+300	1.300
Cx 1	WRITE	800	-200	800
Cx 2	WRITE	1.300	+300	1.300

Problemas de Compartilhamento

- Problema 2
  - Compartilhamento de memória
  - Processos A e B executando um comando de atribuição.

```
Processo A
X := X + 1;
```

```
Processo B
X := X – 1;
```

Problemas de Compartilhamento

- Inicialmente x = 2, qual será o valor de x após a execução dos processos?

```
Processo A
X := X + 1;
```

```
Processo B
X := X – 1;
```

Problemas de Compartilhamento

Processo A	Processo B
LOAD x, Ra	LOAD x, Rb
ADD 1, Ra	SUB 1, Rb
STORE Ra, x	STORE Rb, x

Processo	*Comando*	*X*	*Ra*	*Rb*
A	LOAD X,Ra	2	2	*

Problemas de Compartilhamento

Processo A	Processo B
LOAD x, Ra	LOAD x, Rb
ADD 1, Ra	SUB 1, Rb
STORE Ra, x	STORE Rb, x

Processo	*Comando*	*X*	*Ra*	*Rb*
A	LOAD X,Ra	2	2	*
A	ADD 1,Ra	2	3	*

Problemas de Compartilhamento

Processo A	Processo B
LOAD x, Ra	LOAD x, Rb
ADD 1, Ra	SUB 1, Rb
STORE Ra, x	STORE Rb, x

Processo	*Comando*	*X*	*Ra*	*Rb*
A	LOAD X,Ra	2	2	*
A	ADD 1,Ra	2	3	*
B	LOAD X,Rb	2	*	2

Problemas de Compartilhamento

Processo A	Processo B
LOAD x, Ra	LOAD x, Rb
ADD 1, Ra	SUB 1, Rb
STORE Ra, x	STORE Rb, x

Processo	*Comando*	*X*	*Ra*	*Rb*
A	LOAD X,Ra	2	2	*
A	ADD 1,Ra	2	3	*
B	LOAD X,Rb	2	*	2
B	SUB 1,Rb	2	*	1

Problemas de Compartilhamento

Processo A	Processo B
LOAD x, Ra	LOAD x, Rb
ADD 1, Ra	SUB 1, Rb
STORE Ra, x	STORE Rb, x

Processo	*Comando*	*X*	*Ra*	*Rb*
A	LOAD X,Ra	2	2	*
A	ADD 1,Ra	2	3	*
B	LOAD X,Rb	2	*	2
B	SUB 1,Rb	2	*	1
A	STORE Ra, X	3	3	*

Problemas de Compartilhamento

Processo A	Processo B
LOAD x, Ra	LOAD x, Rb
ADD 1, Ra	SUB 1, Rb
STORE Ra, x	STORE Rb, x

Processo	*Comando*	*X*	*Ra*	*Rb*
A	LOAD X,Ra	2	2	*
A	ADD 1,Ra	2	3	*
B	LOAD X,Rb	2	*	2
B	SUB 1,Rb	2	*	1
A	STORE Ra, X	3	3	*
B	STORE Rb, X	1	*	1

Exclusão Mútua

- Solução simples:
  - Evitar que determinados trechos críticos dos programas sejam executados ao mesmo tempo por mais de um processo.
  - Esses trechos são chamados de *região crítica*
  - O impedimento de acesso simultâneo à região crítica é denominado *Exclusão Mútua*

Exclusão Mútua

- Protocolos de acesso
  - Necessários para controlar entrada e saída da região crítica.

```
BEGIN
   ...
   Entra_Regiao_Critica; /* Procolo de entrada */
   Regiao_Critica();
   Sai_Regiao_Critica;   /* Procolo de saída   */
   ...
END.
```

Exclusão Mútua

- Espera indefinida *(Starvation)*
  - Processo nunca consegue acessar sua região crítica.
  - Causas
    - Escalonamento aleatório
    - Escalonamento por prioridade
  - Solução
    - Usar fila de processos.
    - Marcar n.º de vezes que o processo foi ignorado.

Exclusão Mútua

- Diversas soluções para implementação.
  - Soluções de Hardware
    - Mecanismos de hardware impedem o acesso múltiplo à região crítica.
  - Soluções de Software
    - Mecanismos de software (programas) impedem o acesso múltipo à região crítica.

Exclusão Mútua (Soluções de Hardware)

- Desabilitar interrupções
  - Solução mais simples
  - Inibir interrupções ➔ Inibir *troca de contexto*
  - Limitações:
    - Prejudica concorrência
    - Possibilita inibição infinita
    - Demora na propagação em sistemas com vários processadores

Exclusão Mútua (Soluções de Hardware)

- Desabilitar interrupções

```
BEGIN
   ...
   Desabilita_Interrupcoes;
   Regiao_Critica();
   Habilita_Interrupcoes;
   ...
END.
```

Exclusão Mútua (Soluções de Hardware)

- Test-and-Set
  - Instrução atômica implementada em diversos processadores
  - Evita a manipulação de uma variável por dois processos ao mesmo tempo

```
Test-and-Set(X, Y);
```

  - Valor lógico de Y é copiado para X e
  - Y recebe o valor lógico verdadeiro

Test-and-Set

```
PROGRAM Test_and_Set;
 VAR Bloqueio: BOOLEAN;


PROCEDURE Processo_A;
  VAR Pode_A: BOOLEAN;
BEGIN
  REPEAT
    Pode_A := True;
    WHILE (Pode_A) DO
      Test_and_Set(Pode_A, Bloqueio);
    Regiao_Critica_A;
    Bloqueio := False;
  UNTIL False;
END;
```

Test-and-Set

```
PROGRAM Test_and_Set;
 VAR Bloqueio: BOOLEAN;


PROCEDURE Processo_B;
  VAR Pode_B: BOOLEAN;
BEGIN
  REPEAT
    Pode_B := True;
    WHILE (Pode_B) DO
      Test_and_Set(Pode_B, Bloqueio);
    Regiao_Critica_B;
    Bloqueio := False;
  UNTIL False;
END;
```

Test-and-Set

- Trecho do programa principal

```
BEGIN
  Bloqueio := False;
  PARBEGIN
    Processo_A;
    Processo_B;
  PAREND;
END.
```

Test-and-Set

- Problemas
  - *Starvation*
    - Escalonamento arbitrário de processos

Exclusão Mútua (Soluções de Software)

- Diversas soluções existentes
- Abordagem evolutiva
  - Primeiros algoritmos consideravam apenas para 2 processos.
  - Evolução dos algoritmos permitiu controle de N processos

Exclusão Mútua (Soluções de Software)

- Primeiro Algoritmo
  - Controla 2 processos: A e B
  - Uso de variável de bloqueio
    - Variável global define quem deve executar a *região crítica*

Primeiro Algoritmo

```
PROGRAM Algoritmo_1;
VAR Vez: CHAR;
```

```
BEGIN
  Vez := 'A';
  PARBEGIN
    Processo_A;
    Processo_B;
  PAREND;
END.
```

```
PROCEDURE Processo_A;
 BEGIN
  REPEAT
    WHILE (Vez = 'B') DO;
    Regiao_Critica_A;
    Vez := 'B';
    Processamento_A;
  Until False;
END;
```

```
PROCEDURE Processo_B;
 BEGIN
  REPEAT
    WHILE (Vez = 'A') DO;
    Regiao_Critica_B;
    Vez := 'A';
    Processamento_B;
  Until False;
END;
```

Primeiro Algoritmo

- Problemas
  - Processo pode “parar” durante a execução de sua *região crítica*.
    - Impede outro processo de executar
  - Execução necessariamente alternada
    - Processos com muita necessidade de execução ficam prejudicados
    - Execução de *região crítica* fica limitada pela velocidade do processo mais lento

Primeiro Algoritmo

- Limitação da velocidade de execução

Processo_A	Processo_B	Vez
`While (Vez = 'B')`	`While (Vez = 'A')`	A
`Regiao_Critica_A`	`While (Vez = 'A')`	A
`Vez := 'B'`	`While (Vez = 'A')`	B
`Processamento_A`	`Regiao_Critica_B`	B
`Processamento_A`	`Vez := 'A'`	A
`Processamento_A`	`Processamento_B`	A
`Processamento_A`	`While (Vez = 'A')`	A
`Processamento_A`	`While (Vez = 'A')`	A

Exclusão Mútua (Soluções de Software)

- Segundo Algoritmo
  - 1º algoritmo usa apenas uma variável global
    - Controle de quem tem a vez
  - 2º algoritmo usa uma variável para cada processo.
    - Controle de quem está usando no momento

Segundo Algoritmo

```
BEGIN
  CA := false;
  CB := false;
  PARBEGIN
    Processo_A;
    Processo_B;
  PAREND;
END.
```

```
PROGRAM Algoritmo_2;
VAR CA, CB: BOOLEAN;
```

```
PROCEDURE Processo_A;
BEGIN
 REPEAT
   WHILE (CB) DO;
   CA := True;
   Regiao_Critica_A;
   CA := False;
   Processamento_A;
 Until False;
END;
```

```
PROCEDURE Processo_B;
BEGIN
 REPEAT
   WHILE (CA) DO;
   CB := True;
   Regiao_Critica_B;
   CB := False;
   Processamento_B;
 Until False;
END;
```

Segundo Algoritmo

- Qual(is) o(s) problema(s) do segundo algoritmo?

Terceiro Algoritmo

```
PROGRAM Algoritmo_3;
  VAR CA, CB: BOOLEAN;

 PROCEDURE Processo_A;
 BEGIN
   REPEAT
     CA := True;
     WHILE (CB) DO;
     Regiao_Critica_A;
     CA := False;
     Processamento_A;
   UNTIL False;
 END;
```

```
PROCEDURE Processo_B;
BEGIN
  REPEAT
    CB := True;
    WHILE (CA) DO;
    Regiao_Critica_B;
    CB := False;
    Processamento_B;
  UNTIL False;
END;
```

Terceiro Algoritmo

- Características:
  - Garante exclusão mútua;
  - Possibilita bloqueio indefinido de ambos os processos.
  - Os dois processos alteram as variáveis CA e CB antes da execução da instrução *While*.

Quarto Algoritmo

```
PROGRAM Algoritmo_4;
  VAR CA, CB: BOOLEAN;

PROCEDURE Processo_A;
BEGIN
  REPEAT
    CA := True;
    WHILE (CB) DO
    BEGIN
      CA := False;
      Tempo_Aleatorio;
      CA := True;
    END;
    Regiao_Critica_A;
    CA := False;
  UNTIL False;
END;
```

```
PROCEDURE Processo_B;
BEGIN
  REPEAT
    CB := True;
    WHILE (CA) DO
    BEGIN
      CB := False;
      Tempo_Aleatorio;
      CB := True;
    END;
    Regiao_Critica_B;
    CB := False;
  UNTIL False;
END;
```

Quarto Algoritmo

```
PROCEDURE Tempo_Aleatorio;
BEGIN
  {pequeno intervalo de tempo aleatório}
END;


BEGIN
  CA := False;
  CB := False;
  PARBEGIN
    Processo_A;
    Processo_B;
  PAREND;
END.
```

Quarto Algoritmo

- Características:
  - Garante exclusão mútua;
  - Não gera bloqueio simultâneo de processos;
  - Tempos aleatórios muito próximos e concorrência
    - Ambos processos alteram o valor de CA e CB para falso antes do término do loop.
    - Nenhum dos dois processos conseguirá executar sua região crítica.

Algoritmo de Peterson

```
PROGRAM Algoritmo_Peterson;
  VAR CA, CB: BOOLEAN;

PROCEDURE Processo_A;
BEGIN
  REPEAT
    CA := True;
    Vez := 'B';
    WHILE (CB and Vez='B')DO;
    Regiao_Critica_A;
    CA := False;
    Processamento_A;
  UNTIL False;
END;
```

```
PROCEDURE Processo_B;
BEGIN
  REPEAT
    CB := True;
    Vez := 'A';
    WHILE (CA and Vez='A')DO;
    Regiao_Critica_B;
    CB := False;
    Processamento_B;
  UNTIL False;
END;
```

Algoritmo de Peterson

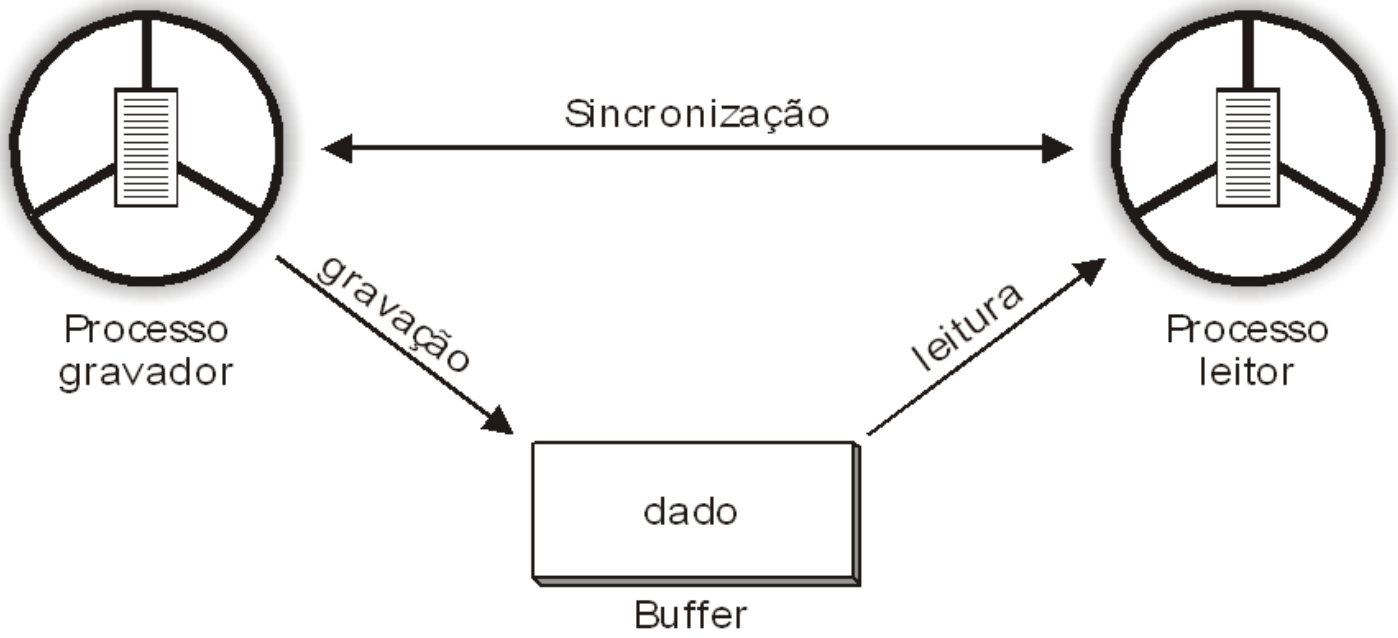
```
BEGIN
   CA := False;
   CB := False;
   PARBEGIN
      Processo_A;
      Processo_B;
   PAREND;
END.
```

Algoritmo de Peterson

- Características:
  - Variáveis CA e CB indicam desejo de um processo de entrar em sua região crítica;
  - Variável Vez resolve conflitos gerados pela concorrência;
  - Garante exclusão mútua;
  - Bloqueio indefinido não ocorre:
    - Variável Vez sempre permite a continuidade da execução de um dos processos.

Sincronização Condicional

- Acesso ao recurso compartilhado exige a sincronização de processos vinculada a uma condição de acesso.
  - Ex:gravação e leitura em um buffer.



Sincronização Condicional

- Problema do produtor/consumidor

```
PROGRAM Produtor_Consumidor_1;
  CONST TamBuf = (* Tam. qualquer *);
  TYPE Tipo_Dado = (* Tipo qualquer *);
VAR
    Buffer: ARRAY [1..TamBuf] OF
    Tipo_Dado;
    Dado: Tipo_Dado;
    Cont: 0..TamBuf;
```

Produtor / Consumidor

```
PROCEDURE Produtor;
BEGIN
  REPEAT
    Produz_Dado (Dado);
    WHILE (Cont = TamBuf) DO;
    Grava_Buffer (Dado, Cont);
  UNTIL False;
END;

PROCEDURE Consumidor;
BEGIN
  REPEAT
    WHILE (Cont = 0) DO;
    Le_Buffer (Dado);
    Consome_Dado (Dado, Cont);
  UNTIL False;
END;
```

```
BEGIN
  Cont := 0;
  PARBEGIN
    Produtor;
    Consumidor;
  PAREND;
END.
```

Produtor / Consumidor

- Resolve sincronização condicional;
- Possui o problema de espera ocupada.

Semáforos

- Proposto por Dijkstra em 1965;
- Permite implementar exclusão mútua e sincronização condicional entre processos;
- Maioria das linguagens disponibiliza rotinas para uso de semáforos.

Semáforos

- Um semáforo é uma variável inteira não negativa que só pode ser manipulada por duas instruções atômicas: DOWN e UP.
  - Instrução UP: incrementa em uma unidade o valor do semáforo
  - Instrução DOWN decrementa em uma unidade o valor do semáforo
    - Caso valor seja zero processo entra em espera.

Semáforos

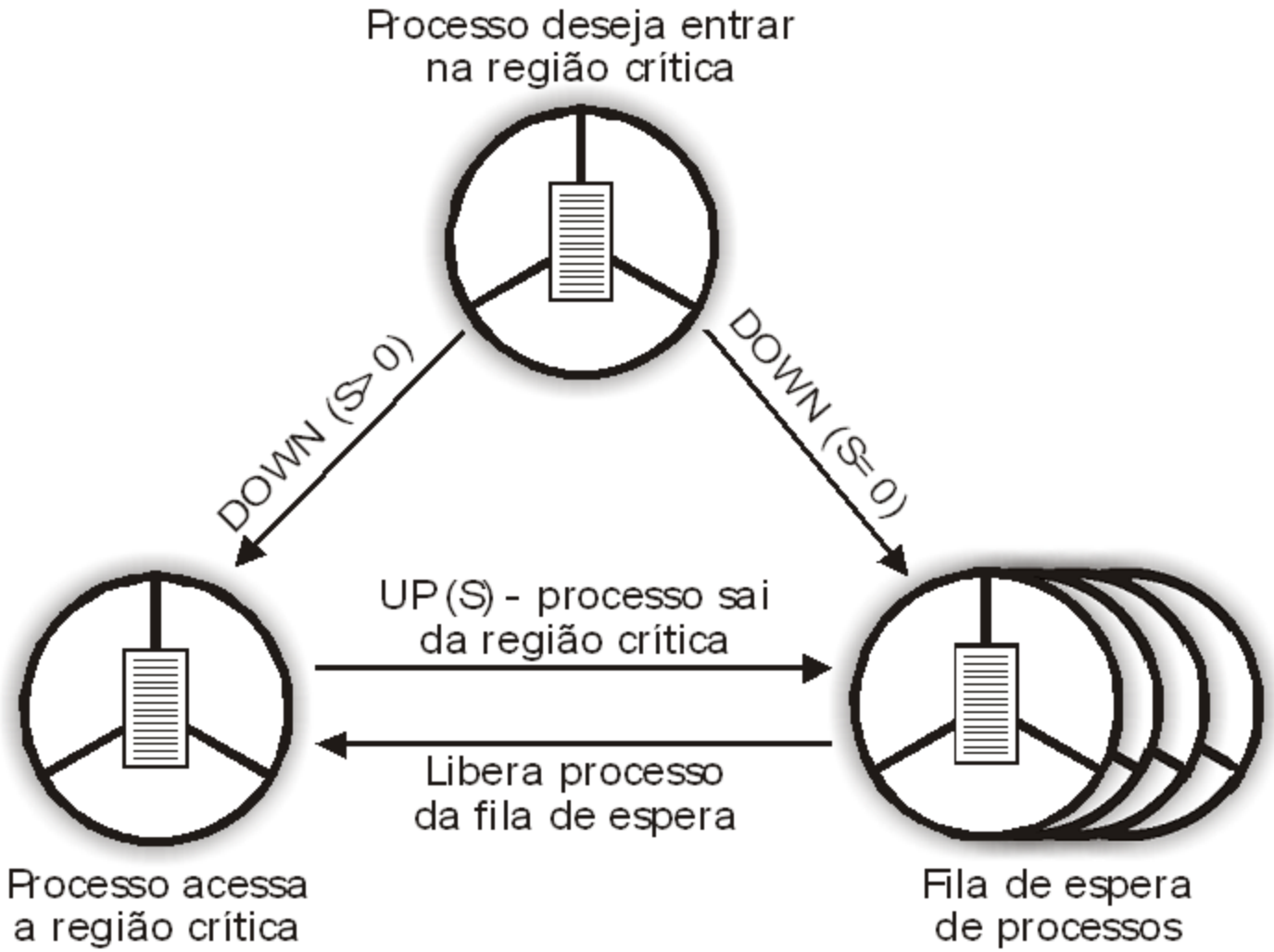
- Classificação
  - Binários: só podem assumir os valores 0 e 1
  - Contadores: podem assumir qualquer valor inteiro maior ou igual a zero.

Semáforos – Exclusão Mútua

- Implementada utilizando semáforo binário.
- Instruções UP e DOWN funcionam como protocolos de entrada e saída.
  - Semáforo igual a 1 → recurso liberado
  - Semáforo igual a 0 → recurso ocupado

- Resolve problema de *espera ocupada*

Semáforos – Exclusão Mútua



Semáforos – Exclusão Mútua

```
PROGRAM Semaforo_1;
VAR s: Semaforo;

BEGIN
  s := 1;
  PARBEGIN
    Processo_A;
    Processo_B;
  PAREND;
END.
```

```
PROCEDURE Processo_A;
BEGIN
  REPEAT
    DOWN (s);
    Regiao_Critica_A;
    UP(s);
  UNTIL False;
END;
```

```
PROCEDURE Processo_B;
BEGIN
  REPEAT
    DOWN (s);
    Regiao_Critica_B;
    UP(s);
  UNTIL False;
END;
```

Semáforos – Exclusão Mútua

Processo_A	Processo_B	s	Pendente
REPEAT	REPEAT	1	*
DOWN(S)	REPEAT	0	*
Reg_Critica_A	DOWN(S)	0	Processo_B
UP(S)	DOWN(S)	1	Processo_B
REPEAT	Reg_Critica_B	0	*

Semáforos – Sincronização Condicional (Ex.1)

```
PROGRAM Semaforo_2;
  VAR Evento: Semaforo;
```

```
BEGIN
  Evento := 0;
  PARBEGIN
    Solicita_Leitura;
    Le_Dados;
  PAREND;
END.
```

```
PROCEDURE Solicita_Leitura;
BEGIN
  DOWN(Evento);
END;
```

```
PROCEDURE Le_Dados;
BEGIN
  ...
  UP(Evento);
END;
```

Semáforos – Sincronização Condicional (Ex. 2)

- Produtor / Consumidor

```
PROGRAM Produtor_Consumidor_2;
  CONST TamBuf = 2;
  TYPE Tipo_Dado = (* Tipo qualquer *);
  VAR
    Vazio, Cheio, Mutex: Semaforo;
    Buffer: ARRAY [1..TamBuf] OF Tipo_Dado;
    Dado: Tipo_Dado;
```

Semáforos – Sincronização Condicional (Ex. 2)

```
PROCEDURE Produtor;
BEGIN
  REPEAT
    Produz_Dado (Dado);
    DOWN(Vazio);
    DOWN(Mutex);
    Grava_Buffer (Dado,Buffer);
    UP(Mutex);
    UP(Cheio);
    UNTIL False;
END;
```

```
PROCEDURE Consumidor;
BEGIN
  REPEAT
    DOWN(Cheio);
    DOWN(Mutex);
    Le_Dado (Dado,Buffer);
    UP(Mutex);
    UP(Vazio);
    Consome_Dado(Dado);
  UNTIL False;
END;
```

Semáforos – Sincronização Condicional (Ex. 2)

- Produtor / Consumidor

```
BEGIN
  Vazio := TamBuf;
  Cheio := 0;
  Mutex := 1;
  PARBEGIN
    Produtor;
    Consumidor;
  PAREND;
END.
```

Semáforos – Sincronização Condicional (Ex. 2)

Produtor	Consumidor	Vazio	Cheio	Mutex	Pendente
*	*	2	0	1	*
*	DOWN(Cheio)	2	0	1	Consumidor
DOWN(Vazio)	DOWN(Cheio)	1	0	1	Consumidor
DOWN(Mutex)	DOWN(Cheio)	1	0	0	Consumidor
Grava_Buffer	DOWN(Cheio)	1	0	0	Consumidor
UP(Mutex)	DOWN(Cheio)	1	0	1	Consumidor
UP(Cheio)	DOWN(Cheio)	1	1	1	*
UP(Cheio)	DOWN(Cheio)	1	0	1	*
UP(Cheio)	DOWN(Mutex)	1	0	0	*
UP(Cheio)	Lê(Dado)	1	0	0	*
UP(Cheio)	UP(Mutex)	1	0	1	*
UP(Cheio)	UP(Vazio)	2	0	1	*

Semáforos – Sincronização Condicional

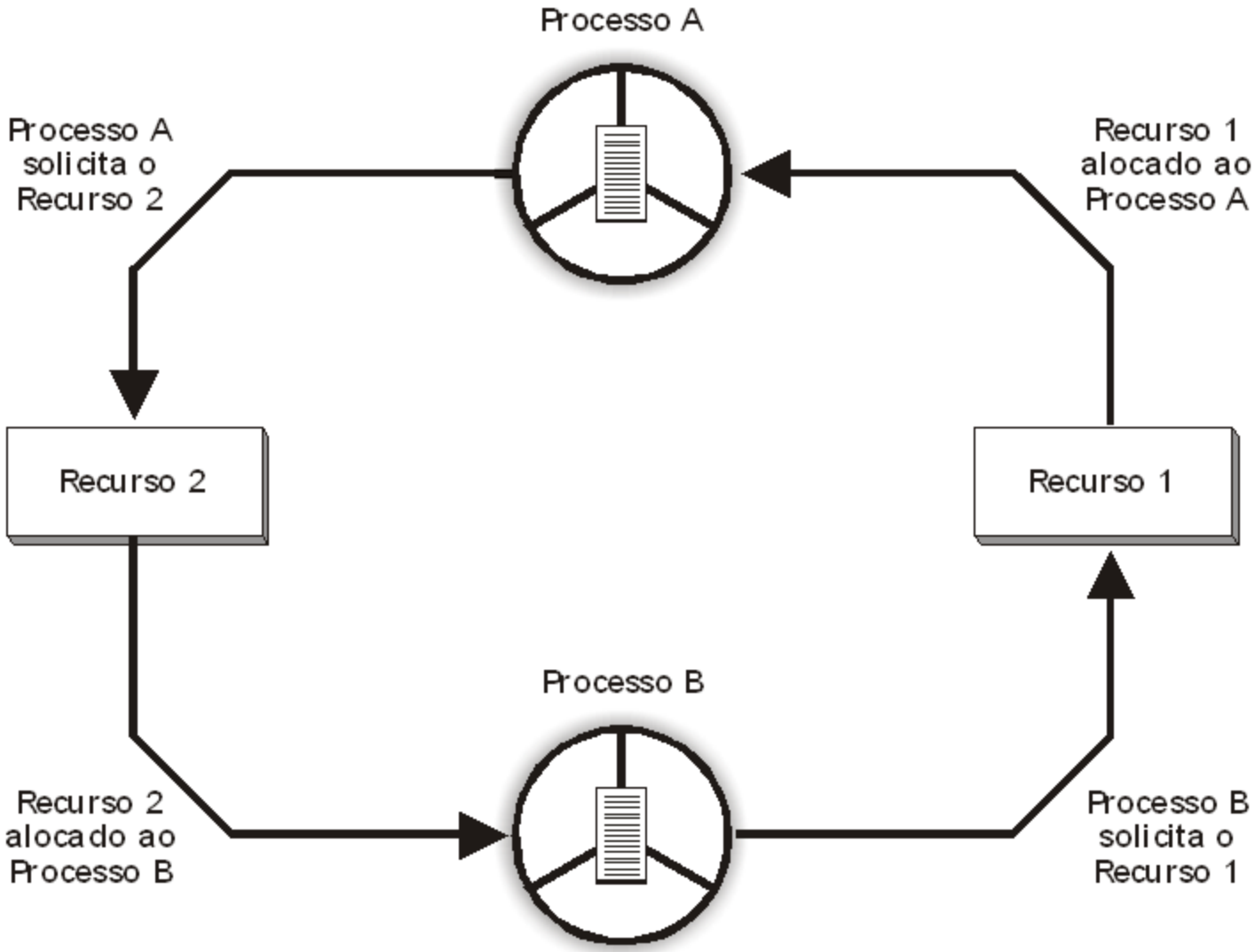
- Semáforos Contadores
  - Úteis em problemas de sincronização condicional;
  - Semáforo é inicializado com o número de recursos disponíveis;
    - Down → Aloca recurso
    - UP → Libera recurso
    - Semáforo = zero → processo espera por recurso

Deadlock

- Processo aguarda por um recurso que nunca estará disponível.
  - Ex: Dispositivos, Arquivos, Registros, etc.

- + Compartilhamento ➔ + Deadlock

Deadlock


Processo A
Processo A
solicita o
Recurso 2
Recurso 1
alocado ao
Processo A
Recurso 2
Recurso 1
Processo B
Recurso 2
alocado ao
Processo B
Processo B
solicita o
Recurso 1

Deadlock

- Causas
  - Exclusão mútua
  - Espera por recurso
  - Não-preempção
  - Espera circular

Deadlock - Prevenção

- Não permitir uma de suas causas
  - Exclusão mútua:
  - Espera por recurso:
    - Solução: Alocar todos os recursos necessários no início [desperdício; nº de recursos; *Starvation*]
  - Não-preempção
    - Solução: Permitir retirar recurso de outro processo [geração de erros; *Starvation*]
  - Espera Circular
    - Solução: Processo só pode alocar um recurso por vez [extremamente limitador]

Deadlock - Prevenção

- Algoritmo de Dijkstra (1965)
  - Processos informam o número máximo de cada tipo de recurso.
  - Sistema calcula relação de recursos alocados/disponíveis e controla alocação.
  - Limitações:
    - Exige número fixo de processos ativos
    - Exige número fixo de recursos no sistema

Deadlock - Detecção

- Prevenir nem sempre é possível
  - Sistemas normalmente tentam então detectar deadlocks no sistema.
  - Uso de estrutura de dados sobre
    - Cada recurso do sistema
    - Para qual processo cada recurso esta alocado
    - Qual processo está aguardando e por qual recurso
  - Algoritmos se baseiam na detecção de espera circular

Deadlock - Correção

- Eliminar um ou mais processos envolvidos;
  - Necessidade de realizar *rollback*
  - Processo escolhido segundo critério do SO:
    - Aleatoriamente
    - Prioridade

Exercícios

1. Defina o que é uma aplicação concorrente e dê exemplo de sua utilização.

2. Considere uma aplicação que utilize uma matriz na memória principal para a comunicação entre vários processos concorrentes. Que tipo de problema pode ocorrer quando dois ou mais processos acessam uma mesma posição da matriz?

3. O que é exclusão mútua e como é implementada?

4. Qual o problema com a solução que desabilita as interrupções para implementar a exclusão mútua

Exercícios

5. Explique o que é sincronização condicional e dê um exemplo de sua utilização

6. Explique o que são semáforos e dê 2 exemplos de sua utilização (um para exclusão mútua e outro para sincronização condicional).

7. O que é espera Ocupada e qual o seu problema?

Exercícios (Threads)

1. Quais os problemas de aplicações concorrentes desenvolvidas em ambientes monothread?

2. Quais as vantagens e desvantagens do compartilhamento do espaço de endereçamento entre threads de um mesmo processo?

3. Quais os benefícios do uso de threads em ambientes cliente-servidor

4. Dê exemplos do uso de threads no desenvolvimento de aplicativos como editores de texto e planilhas eletrônicas?